Build, Augment and Destroy, Universally

Neil Ghani[1] and Tarmo Uustalu[2] and Varmo Vene[3]

[1] Dept. of Math. and Comp. Sci., University of Leicester
University Road, Leicester, LE1 7RH, N.Ghani@mcs.le.ac.uk
[2] Institute of Cybernetics, Tallinn Univ. of Technology
Akadeemia tee 21, EE-12618 Tallinn, Estonia, tarmo@cs.ioc.ee
[3] Dept. of Computer Science, University of Tartu
Liivi 2, EE-50409 Tartu, Estonia, varmo@cs.ut.ee

Abstract. We give a semantic footing to the FOLD/BUILD syntax of programming with inductive types, covering shortcut deforestation, based on a universal property. Specifically, we give a semantics for inductive types based on limits of algebra structure forgetting functors and show that it is equivalent to the usual initial algebra semantics. We also give a similar semantic account of the AUGMENT generalization of BUILD and of the UNFOLD/DESTROY syntax of coinductive types.

1 Introduction

The beauty of the IN/FOLD and OUT/UNFOLD syntax for programming with inductive and coinductive types [13, 21] is thanks to the fact that it is based directly on the semantic interpretation of such types as initial algebras and final coalgebras. Apart from the clarity provided by this close match between syntax and semantics, there are practical benefits for the programmer. For example, in the case of inductive types, (i) the constructors IN form the structure map of the initial algebra which, as an isomorphism, supports pattern matching in function definitions; (ii) the mediating algebra map to any other algebra gives rise to a canonical recursion combinator usually called FOLD to aid structured programming; (iii) β-equality (also called computational rules) is derived from the fact that the mediating map is an algebra map; and (iv) η-equality and permutative equality (also called extensionality rules), which provide the correctness for program transformations such as FOLD fusion, are derived from the uniqueness of the mediating map. In essence, all the fundamental properties of an inductive type can be derived systematically from its semantics as an initial algebra. The same can be said about coinductive types and their final coalgebra semantics.

Recently there has been a significant amount of work starting with Gill et al.'s [10] which proposes an alternative approach to programming with inductive types based on using the FOLD-combinator in interaction with a new constructor called BUILD. This is related to deforestation [35] or semantics-preserving program transformation to eliminate intermediate datastructures. Use of intermediate datastructures should in principle be encouraged in functional programming as it gives well-structured programs, but the resulting programs are also inefficient, thus the need for automated deforestation. There exist several deforestation methods, but probably the most successful is Gill's shortcut deforestation where removal of intermediate datastructures is achieved by repeated application of very simple rewrite rules, but which requires that programming with inductive intermediate datastructures is done in terms FOLD and BUILD.

Despite the proliferation of this interesting work described, the semantic foundations of the FOLD/BUILD paradigm are not so clear. Thus, for example, various program transformations have to be derived and justified in a relatively ad-hoc manner. We would rather prefer to be able to automatically derive the essential properties of the FOLD/BUILD style of programming from similar principles to the initial algebra semantics underpinning the IN/FOLD style of programming.

This paper achieves this. Our key insight is that initial algebras are an instance of a more general concept of *universal properties* which are pervasive throughout programming language semantics. Thus we give a characterization of initial algebras via an alternative universal property from which the syntax and equational properties of the FOLD/BUILD paradigm arise in the manner described above. The suitable universal property turns out to be rather simple and, moreover, intuitive. It is that of being the limit of a functor forgetting algebra structure or, intuitively, the largest object with a fuseable fold-like operation.

The benefits of our approach are not just to provide semantic clarity by placing the FOLD/BUILD-style of programming on the same universal footing that other programming language constructions have. Indeed, as with all good semantics, we use our universal characterization of FOLD/BUILD to derive a number of other concrete deliverables. Thus, the key contributions of this paper are:

- We give a categorical foundation to the FOLD/BUILD approach to inductive types which replaces the more ad-hoc foundations in the literature with universal constructions. This extracts the properties that BUILD must have in order validate shortcut deforestation and provides a categorical justification to the encoding of inductive types using type quantification.
- Implicit in this is a proof of the equi-expressiveness of the FOLD/BUILD syntax and the usual IN/FOLD syntax. In particular, the FOLD/BUILD constructions suffice to define the IN/FOLD syntax validating all IN/FOLD axioms.
- The flexibility of this semantic approach is demonstrated by the ease with which we extend it to cover the FOLD/AUGMENT fusion of Gill [11] and the UNFOLD/DESTROY paradigm of programming with coinductive types. (As a matter of fact, helped by the transparency of the framework, we have found that a useful AUGMENT combinator is definable not only for free algebras, but for a far more wide class of parameterized inductive types. For space reasons, we will publish that result separately elsewhere, only giving some hints here.)

The paper, appealing both to functional programmers and semanticists, is organized as follows: Sect. 2 begins by a demonstration of how the classical infrastructure for programming and reasoning with inductive types arises from the initial algebra semantics. It continues with an exposition of our alternative and equivalent semantics where the BUILD combinator is a primitive construction and FOLD/BUILD fusion is axiomatic. This alternative semantics is based on a kind of cones that from the functional programming perspective are polymorphic functions required to meet a non-trivial coherence condition; we conclude the section by showing that it is a condition of strong dinaturality. In Sect. 3, we give a justification of Gill's AUGMENT generalization of BUILD in our framework. Sect. 4 is an exposition of the dual development for the semantics of coinductive types. In Sect. 5, we give a condensed orientation about relating work and, in Sect. 6, we conclude, pointing out a number of directions for future work.

From the reader, we assume only the most basic definitions from category theory, in particular those of category, functor and natural transformations. Familiarity with limits and colimits will be helpful, but they are defined in the text. We work with a base category about which we make no or very mild assumptions, but the concrete examples to keep in mind are categories most relevant for programming semantics, like **Set** and **CPO**. Throughout the text, we use Haskell for examples, but this is purely illustrative; we are not discussing the semantics of Haskell or any particular language in this paper.

2 Inductive types and build

2.1 Inductive types as initial algebras

The customary structured approach to list programming equips the type `[a]` of lists over `a` with constructors `[]` and `(:)` (for nil and cons) and a destructor `foldr`. In Haskell, this is accomplished as follows:

```
data [a] = [] | a : [a]

foldr :: (a -> x -> x) -> x -> [a] -> x
foldr c n [] = n
foldr c n (a : as) = c a (foldr c n as)
```

The syntax thus outlined derives directly from the categorical semantics of lists over type A as a chosen initial algebra of the functor $1 + A \times -$.

Algebras and the property of an algebra being initial are defined as follows:

Definition 1 ((initial) algebra). *Let $\mathcal{C}$ be a category and $F : \mathcal{C} \to \mathcal{C}$ be a functor. An F-*algebra *is an object X in $\mathcal{C}$ together with a map $\varphi : FX \to X$ in $\mathcal{C}$. An F-*algebra map *$(X, \varphi) \to (Y, \psi)$ is a map $f : X \to Y$ such that $f \circ \varphi = \psi \circ F f$. An* initial F-algebra *is an initial object in the category F-**alg** of F-algebras, i.e., an F-algebra with a unique map from it to any F-algebra.*

We agree to denote a chosen initial F-algebra by $(\mu F, \mathsf{in}_F)$, and the unique map from it to an F-algebra (X, φ) by $\mathsf{fold}_{F,X}\varphi$.

By the definition, an algebra for the functor $1 + A \times -$ is an object X and a map $\varphi : 1 + A \times X \to X$. The latter is of course equivalent to a pair of maps $\varphi_0 : 1 \to X$ and $\varphi_1 : A \times X \to X$ giving interpretations of `[]` and `(:)`. Maps from an algebra (X, φ) to an algebra (Y, ψ) are just maps from X to Y which preserve the interpretations of `[]` and `(:)` in the algebras. This is of course exactly what a *model* and *model homomorphism* are taken to be in universal algebra. We are interested in a specific algebra, namely the one where `[]` is interpreted as nil and `(:)` as the cons operation (of the given implementation of lists). This means that lists over A are really modelled by an *initial algebra* of $1 + A \times -$.

Note how characterizing lists as a chosen initial algebra provides all the syntax and equalities we need to program and reason with: i) the structure map of the initial algebra provides the constructors nil and cons which, taken together, form an isomorphism, supporting pattern matching:

$$\mathsf{in}_F : F(\mu F) \to \mu F$$

ii) the mediating map provides the FOLD-combinator `foldr`:

$$\frac{(X, \varphi) \in F\text{-}\mathbf{alg}}{\mathsf{fold}_{F,X}\varphi : \mu F \to X}$$

iii) that the FOLD combinator constructs an algebra map gives the equation

$$\frac{(X, \varphi) \in F\text{-}\mathbf{alg}}{\mathsf{fold}_{F,X}\varphi \circ \mathsf{in}_F = \varphi \circ F\mathsf{fold}_{F,X}\varphi}$$

which defines the computational rules of β-equality or FOLD cancellation; and iv) the uniqueness of the mediating map provides the following extensionality rules

$$\mathsf{fold}_{F,F(\mu F)}\mathsf{in}_F = \mathsf{id}_{\mu F} \qquad \frac{f : (X, \varphi) \to (Y, \psi) \in F\text{-}\mathbf{alg}}{f \circ \mathsf{fold}_{F,X}\varphi = \mathsf{fold}_{F,Y}\psi}$$

The first rule is often taken to be η-equality "lite" while the second rule of permutative conversion provides the soundness of FOLD fusion. As with all initial objects, initial algebras are defined up to isomorphism, which means, e.g., that any initial algebra of $1+A\times-$ will be isomorphic to the type of lists over A. Thus the initial algebra semantics defines our list type up to isomorphism, which really means that we can program with it independently of how it is implemented.

2.2 Universal constructions, limits

That so much of the infrastructure to program with lists and inductive types in general is explained by the single concept of an initial algebra is very elegant. Initial algebras are instances of the more general concept of universal constructions which arise in numerous other places in the semantics of programming languages. We will not give the definition of a universal construction here; this can be found in any standard reference, such as [20]. Informally, a universal property states that, for some objects and maps which satisfy some properties, there is a unique map (called the mediating map) between this structure and any other structure satisfying the same properties. Universal properties define structures up to isomorphism. Other examples are, e.g., (categorical) products, which model product types, and general limits, which we will use in this paper.

Products are an instructive familiar example where again the universal property provides all the information we need to program and reason with product types. The fact that the product (final span) is a span gives the projections while pairing arises as the mediating map from any other span. The fact that pairing is a map of spans is precisely β-equality while the uniqueness of the mediating map is the usual η-equality. We can see that all universal properties have an implicit syntax with associated computational and extensionality rules.

We will shortly need the concept of a limit and record here the definition.

Definition 2 (limit). *A* cone *for a functor $J : \mathcal{C} \to \mathcal{D}$ is a pair (C, γ) consisting of an object C in $\mathcal{D}$ and, for each object X in $\mathcal{C}$, a map $\gamma_X : C \to JX$ in $\mathcal{D}$ such that, for every map $f : X \to Y$ in $\mathcal{C}$, we have $Jf \circ \gamma_X = \gamma_Y$. A* map of cones *$(C, \gamma) \to (D, \delta)$ is a map $h : C \to D$ in $\mathcal{D}$ such that, for each object X in $\mathcal{C}$, we have $\delta_X \circ h = \gamma_X$. A* limit *of J is a final object in the category J-**cone** of J-cones.*

2.3 Building build

The initial algebra semantics based syntax for inductive types is elegant and useful in many ways, but the structure it imposes on programs leads often to inefficiency.

Consider the task of programming the sum of the squares of a list of integers. The following is a modular and well-structured program:

```
sumSq m = sum (map square [1..m])
```

Unfortunately, it uses two intermediate datastructures, which of course implies inefficiency. The methodology of shortcut deforestation [10] proposes reprogramming summation, mapping and list generation in terms of `foldr` and a new combinator `build` as follows:

```
build :: (forall x. (a -> x -> x) -> x -> x) -> [a]
build theta = theta (:) []

sum = foldr (+) 0
map f xs = build (\ c n -> foldr (\ x ys -> f x `c` ys) n xs)
upto i1 i2 = build (\ c n -> upto' c n i1 i2)
upto' c n i1 i2 = if i1 > i2 then n else i1 `c` upto' c n (i1 + 1) i2
```

The point of `build` is to abstract over the constructor occurrences in a list to provide `foldr` with direct access to their positions which motivates the `foldr/build` fusion rule

```
foldr c n (build theta) == theta c n
```

Applying this rewrite rule twice, we get a monolithic (and thus not so programmer-friendly) but efficient version:

```
sum . map square
== \ xs -> foldr (+) 0
             (build (\ c n -> foldr (\ x ys -> square x `c` ys) n xs))
== foldr (\ x ys -> square x + ys) 0

foldr (\ x ys -> square x + ys) 0 (upto 1 m)
== foldr (\ x ys -> square x + ys) 0 (build (\ c n -> upto' c n 1 m))
== upto' (\ x ys -> square x + ys) 0 1 m

sumSq m = sumSq' 1 m
sumSq' i1 i2 = if i1 > i2 then 0 else square i1 + sumSq' (i1 + 1) i2
```

This paper explains shortcut deforestation or FOLD/BUILD fusion via an alternative semantics of inductive types which matches the FOLD/BUILD rather than the IN/FOLD syntax but is still equivalent to the initial algebra semantics.

Given a category $\mathcal{C}$ and functor $F : \mathcal{C} \to \mathcal{C}$, let us write $U_F : F\text{-}\mathbf{alg} \to \mathcal{C}$ for the functor forgetting F-algebra structure, i.e., the functor which maps an algebra (X, φ) to X and an F-algebra map $f : (X, \varphi) \to (Y, \psi)$ to $f : X \to Y$.

Our alternative semantics for the inductive type given by a functor F will be the limit of U_F. Let us spell out what U_F-cones are and what a U_F-limit is. By Definition 2, a *U_F-cone* is an object C in $\mathcal{C}$ and, for any F-algebra (X, φ), a map $\Theta_X \varphi : C \to X$ in $\mathcal{C}$, such that (*) for any F-algebra map $f : (X, \varphi) \to (Y, \psi)$, we have $f \circ \Theta_X \varphi = \Theta_Y \psi$. A *$U_F$-cone map* $h : (C, \Theta) \to (D, \Xi)$ is a map $h : C \to D$ in $\mathcal{C}$ such that, for any F-algebra (X, φ), we have $\Xi_X \varphi \circ h = \Theta_X \varphi$. A *$U_F$-limit* is a U_F-cone to which there is a unique map from any other U_F-cone. Let us write $(\mu^* F, \mathsf{fold}^*_F)$ for a chosen U_F-limit and $\mathsf{build}^*_{F,C} \Theta$ for the mediating map from (C, Θ), hinting towards the propositions we will present in a moment. Intuitively, a U_F-cone is an object with an operation which types as a fold operation from that object and is fuseable; the U_F-limit is the greatest such object.

The general idea of deriving syntax from universal properties suggests that we may consider introducing a type $\mu^* F$ and a destructor fold^*_F and constructor build^*_F with typing rules

$$\frac{(X, \varphi) \in F\text{-}\mathbf{alg}}{\mathsf{fold}^*_{F,X} \varphi : \mu^* F \to X} \quad \frac{f : (X, \varphi) \to (Y, \psi) \in F\text{-}\mathbf{alg}}{f \circ \mathsf{fold}^*_{F,X} \varphi = \mathsf{fold}^*_{F,Y} \psi} \quad \frac{(C, \Theta) \in U_F\text{-}\mathbf{cone}}{\mathsf{build}^*_{F,C} \Theta : C \to \mu^* F}$$

As the β-conversion rule, we get

$$\frac{(C, \Theta) \in U_F\text{-}\mathbf{cone} \quad (X, \varphi) \in F\text{-}\mathbf{alg}}{\mathsf{fold}^*_{F,X} \varphi \circ \mathsf{build}^*_{F,C} \Theta = \Theta_X \varphi}$$

and the η- and permutative conversion rules are

$$\mathsf{id}_{\mu^* F} = \mathsf{build}^*_{F,\mu^* F} \mathsf{fold}^*_F \qquad \frac{h : (C, \Theta) \to (D, \Xi) \in U_F\text{-}\mathbf{cone}}{\mathsf{build}^*_{F,D} \Xi \circ h = \mathsf{build}^*_{F,C} \Theta}$$

We see that an U_F-limit provides almost exactly the syntax made use of in FOLD/BUILD fusion: if one ignores the coherence condition (*) of an U_F-cone,

fold^*_F and build^*_F type as FOLD and BUILD should and, moreover, the β-conversion rule is the FOLD/BUILD fusion law. The next proposition shows that an initial F-algebra is, in fact, a U_F-limit.

Proposition 1. *Let $\mathcal{C}$ be a category and $F : \mathcal{C} \to \mathcal{C}$ be a functor. If there is an initial F-algebra $(\mu F, \mathsf{in}_F)$, then μF is the vertex of an U_F-limit.*

Proof. We make use of the definition and properties of an initial F-algebra. We begin by observing that $(\mu F, \mathsf{fold}_F)$ is a U_F-cone, since, for any F-algebra map $f : (X, \varphi) \to (Y, \psi)$, we have $f \circ \mathsf{fold}_{F,X}\varphi = \mathsf{fold}_{F,Y}\psi$. We prove that $(\mu F, \mathsf{fold}_F)$ is the final U_F-cone by showing that, given any U_F-cone (C, Θ), the required mediating U_F-cone map is $\mathsf{build}_{F,C}\Theta =_{\mathrm{df}} \Theta_{\mu F}\ \mathsf{in}_F : C \to \mu F$.

To see that $\mathsf{build}_{F,C}\Theta$ is a U_F-cone map from (C, Θ) to $(\mu F, \mathsf{fold}_F)$, we note that, for any F-algebra (X, φ), the map $\mathsf{fold}_{F,X}\varphi : \mu F \to X$ is an F-algebra map $(\mu F, \mathsf{in}_F) \to (X, \varphi)$, and hence, $\mathsf{fold}_{F,X}\varphi \circ \Theta_{\mu F}\ \mathsf{in}_F = \Theta_X\ \varphi$. To see that $\mathsf{build}_{F,C}\Theta$ is unique, consider any U_F-cone map $h : (C, \Theta) \to (\mu F, \mathsf{fold}_F)$. As $(\mu F, \mathsf{in}_F)$ is an F-algebra, we have $\mathsf{fold}_{F,\mu F}\mathsf{in}_F \circ h = \Theta_{\mu F}\ \mathsf{in}_F$. From this, as $\mathsf{fold}_{F,\mu F}\mathsf{in}_F = \mathsf{id}_{\mu F}$, we get $h = \Theta_{\mu F}\ \mathsf{in}_F$ as required. [4] □

On the basis that inductive types are limits of functors forgetting algebra structure, and that the syntax suggested by this universal property includes a combinator exhibiting all that is expected from BUILD (in particular FOLD/BUILD fusion), although posing a coherence condition on its argument, we take this universal property to constitute the specification of BUILD. The usual FOLD/BUILD fusion will be correct, if coherence is automatic for actual invocations of BUILD. We will touch upon this question in the next subsection.

We have shown that, if F has an initial algebra, it has a final U_F-cone, i.e., that the initial algebra modelling of an inductive type is at least as strong as the limit of forgetful functor modelling: from the IN/FOLD-syntax, we can define the FOLD/BUILD-syntax. But, in fact, the reverse is also true, the two notions are equi-expressive.

Proposition 2. *If there is a U_F-limit $(\mu^* F, \mathsf{fold}^*_F)$, then $\mu^* F$ is the carrier of an initial F-algebra.*

Proof. Define, for any F-algebra (X', φ'), a map $\mathsf{infold}^*_{F,X'}\varphi' : F(\mu^* F) \to X'$ by

$$\mathsf{infold}^*_{F,X'}\ \varphi' =_{\mathrm{df}} \varphi' \circ F\ (\mathsf{fold}^*_{F,X'}\ \varphi')$$

For any F-algebra map $f : (X', \varphi') \to (Y', \psi')$, we have

$$\begin{aligned} f \circ \mathsf{infold}^*_{F,X'}\ \varphi' &= f \circ \varphi' \circ F\ (\mathsf{fold}^*_{F,X'}\ \varphi') \\ &= \psi' \circ F\ (f \circ \mathsf{fold}^*_{F,X'}\ \varphi') \\ &= \psi' \circ F\ (\mathsf{fold}^*_{F,Y'}\ \psi') \\ &= \mathsf{infold}^*_{F,Y'}\ \psi' \end{aligned}$$

[4] Shorter: An initial object is a limit of the identity functor, so an initial F-algebra is a $\mathsf{Id}_{F\text{-}\mathbf{alg}}$-limit. By the preservation of limits by right adjoints, its carrier is a U_F-limit.

This tells us that $(F(\mu^* F), \mathsf{infold}^*_F)$ is a U_F-cone.

Define now a map $\mathsf{in}^*_F : F(\mu^* F) \to \mu^* F$ by

$$\mathsf{in}^*_F =_{\mathrm{df}} \mathsf{build}^*_{F,F(\mu^* F)} \mathsf{infold}^*_F$$

Obviously $(\mu^* F, \mathsf{in}^*_F)$ is an F-algebra. We prove that it is initial by showing that, given any F-algebra (X, φ), the map $\mathsf{fold}^*_{F,X}\varphi : \mu^* F \to X$ is the required unique mediating F-algebra map.

To see that $\mathsf{fold}^*_{F,X}\varphi$ is an F-algebra map $(\mu^* F, \mathsf{in}^*_F) \to (X, \varphi)$, we invoke that $(F(\mu^* F), \mathsf{infold}^*_F)$ is a U_F-cone to observe that

$$\begin{aligned}
\mathsf{fold}^*_{F,X}\ \varphi \circ \mathsf{in}^*_F &= \mathsf{fold}^*_{F,X}\ \varphi \circ \mathsf{build}^*_{F,F(\mu^* F)} \mathsf{infold}^*_F \\
&= \mathsf{infold}^*_{F,X}\ \varphi \\
&= \varphi \circ F\ (\mathsf{fold}^*_{F,X}\ \varphi)
\end{aligned}$$

It remains to verify that $\mathsf{fold}^*_{F,X}\varphi$ is unique.

We have just seen that $\mathsf{fold}^*_{F,X'}\ \varphi'$ is an F-algebra map $(\mu^* F, \mathsf{in}^*_F) \to (X', \varphi')$ for any F-algebra (X', φ'), hence $\mathsf{fold}^*_{F,X'}\ \varphi' \circ \mathsf{fold}^*_{F,\mu^* F}\ \mathsf{in}^*_F = \mathsf{fold}^*_{F,X'}\ \varphi'$. It follows that $\mathsf{fold}^*_{F,\mu^* F}\ \mathsf{in}^*_F$ is a U_F-cone map from the U_F-limit $(\mu^* F, \mathsf{fold}^*_F)$ to itself, therefore $\mathsf{fold}^*_{F,\mu^* F}\ \mathsf{in}^*_F = \mathsf{id}_{\mu F}$.

Now consider any F-algebra map $f : (\mu^* F, \mathsf{in}^*_F) \to (X, \varphi)$. We have $f = f \circ \mathsf{fold}^*_{F,\mu^* F}\ \mathsf{in}^*_F = \mathsf{fold}^*_{F,X}\ \varphi$, which completes the uniqueness proof. $\square$

This proposition assures us that the FOLD/BUILD paradigm is no weaker than the customary IN/FOLD. Everything we can program with IN/FOLD, we can also do with FOLD/BUILD. E.g., we may implement natural numbers as follows:

```
data Nat = BuildN (forall x . x -> (x -> x) -> x)

foldN :: x -> (x -> x) -> Nat -> x
foldN z s (BuildN theta) = theta z s
```

Now, the zero and successor constructors and, in fact, all numbers are definable:

```
zeroN :: Nat
zeroN = BuildN (\ z _ -> z)

succN :: Nat -> Nat
succN n = BuildN (\ z s -> s (foldN z s n))

toN :: Int -> Nat
toN n = BuildN (\ z s -> ntimes n s z)

ntimes n f = if n == 0 then id else f . ntimes (n - 1) f
```

We see that what we get are the Church numerals. How FOLD/BUILD fusion for natural numbers works is therefore intuitively very clear: By abstracting out the occurrences of the zero and successor constructors in a numeral, we obtain direct access to their positions, and it is exactly these constructor occurrences that folds replace. More generally, we have constructed a categorical version of the type quantification based encoding of inductive types of [19, 3].

2.4 Strong dinatural transformations

We now turn to the question about the coherence condition of U_F-cones. It is rather well-behaved: we will shortly see that it is a strong dinaturality condition.

Dinatural transformations and strongly dinatural transformations are two generalizations of the concept of natural transformation for mixed-variant functors. Standard dinaturality, as introduced by Dubuc and Street [4], is fairly well known, but, for reference, we record the definition.

Definition 3 (dinaturality). *Let $H, K : \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \mathcal{D}$ be functors. A* dinatural transformation *$\Theta : H \to K$ is a family of maps $\Theta_X : H(X,X) \to K(X,X)$ in $\mathcal{D}$ for all objects X in $\mathcal{C}$ such that, for every map $f : X \to Y$ in $\mathcal{C}$, the following hexagon commutes:*

$$\begin{tikzcd}
 & H(X,X) \arrow[r, "\Theta_X"] & K(X,X) \arrow[dr, "K(X,f)"] & \\
H(Y,X) \arrow[ur, "H(f,X)"] \arrow[dr, "H(Y,f)"'] & & & K(X,Y) \\
 & H(Y,Y) \arrow[r, "\Theta_Y"'] & K(Y,Y) \arrow[ur, "K(f,Y)"'] &
\end{tikzcd}$$

A major deficiency of standard dinatural transformations is that they do not generally compose. Strongly dinatural transformations address and solve exactly this problem. They appeared in Mulry's paper [23], but the exact authorship is unclear: in the beautiful in-depth account [24], they are attributed to Barr, personal communication.

Definition 4 (strong dinaturality). *Let $H, K : \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \mathcal{D}$ be functors. A* strongly dinatural transformation *$\Theta : H \to K$ is a family of maps $\Theta_X : H(X,X) \to K(X,X)$ in $\mathcal{D}$ for all objects X in $\mathcal{C}$ such that, for every map $f : X \to Y$ in $\mathcal{C}$, object W and maps $p_0 : W \to H(X,X)$, $p_1 : W \to H(Y,Y)$ in $\mathcal{D}$, if the square in the following diagram commutes, then so does the hexagon:*

$$\begin{tikzcd}
 & H(X,X) \arrow[rr, "\Theta_X"] \arrow[dr, "H(X,f)"] & & K(X,X) \arrow[dr, "K(X,f)"] & \\
W \arrow[ur, "p_0"] \arrow[dr, "p_1"'] & & H(X,Y) \arrow[r, phantom, "\Rightarrow"] & & K(X,Y) \\
 & H(Y,Y) \arrow[rr, "\Theta_Y"'] \arrow[ur, "H(f,Y)"] & & K(Y,Y) \arrow[ur, "K(f,Y)"'] &
\end{tikzcd}$$

If $\mathcal{D}$ is a category with pullbacks such as, e.g., **Set**, one can equivalently require that, for every map $f : X \to Y$ in $\mathcal{C}$, the outer hexagon of the above diagram commutes for (W, p_0, p_1) the chosen pullback of $H(X,f)$ and $K(f,Y)$.

It is easy to see that every strongly dinatural transformation is also dinatural, but the converse does not hold in general.

We will need strongly dinatural transformations between $H, K : \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \mathcal{D}$ for the special case where $\mathcal{C}$ is locally small (meaning that all homcollections

$\mathsf{Hom}(A, B)$ are sets), $\mathcal{D} = \mathbf{Set}$ and $H = \mathsf{Hom}(F\ -, G\ -) : \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \mathbf{Set}$ for some functors $F, G : \mathcal{C} \to \mathcal{C}$. In this situation, the difference between dinaturality and strong dinaturality is especially apparent. A dinatural transformation $\Theta : H \to K$ is a family of functions $\Theta_X : \mathsf{Hom}(F\ X, G\ X) \to K\ X$ such that, for any maps $f : X \to Y$, $\xi : F\ Y \to G\ X$, $\varphi : F\ X \to G\ X$, $\psi : F\ Y \to G\ Y$, if the two triangles

$$\begin{tikzcd} F\ X \arrow[r, "\varphi"] \arrow[d, "F\ f"'] & G\ X \arrow[d, "G\ f"] \\ F\ Y \arrow[ur, "\xi"] \arrow[r, "\psi"'] & G\ Y \end{tikzcd}$$

commute, then $K\ (X, f)\ (\Theta_X\ \varphi) = K\ (f, Y)\ (\Theta_Y\ \psi)$. For a strongly dinatural transformation, in contrast, the condition is: for any maps $f : X \to Y$, $\varphi : F\ X \to G\ X$, $\psi : F\ Y \to G\ Y$, if the square

$$\begin{tikzcd} F\ X \arrow[r, "\varphi"] \arrow[d, "F\ f"'] & G\ X \arrow[d, "G\ f"] \\ F\ Y \arrow[r, "\psi"'] & G\ Y \end{tikzcd}$$

commutes, then $K\ (X, f)\ (\Theta_X\ \varphi) = K\ (f, Y)\ (\Theta_Y\ \psi)$. Clearly the second one of the two implications is stronger because of its weaker antecedent.

Comparing the unwinding of the definition of strong dinaturality and our ealier unwinding of the definition of limit, we arrive at the following observation.

Proposition 3. *Let $\mathcal{C}$ be a locally small category and $F : \mathcal{C} \to \mathcal{C}$ a functor. A U_F-cone structure with vertex C is the same thing as a strong dinatural transformation from $\mathsf{Hom}(F\ -, -)$ to $\mathsf{Hom}(C, -)$.*

Proof. Both U_F-cone structures and strong dinatural transformations from $\mathsf{Hom}(F-, -)$ to $\mathsf{Hom}(C, -)$ are families of maps $\Theta_X : \mathsf{Hom}(FX, X) \to \mathsf{Hom}(C, X)$ such that, for any maps $f : X \to Y$, $\varphi : FX \to X$, $\psi : FY \to Y$, if $f \circ \varphi = \psi \circ Ff$, then $f \circ \Theta_X \varphi = \Theta_Y \psi$. $\square$

The proposition tells us that the typing rule for BUILD can be rewritten as

$$\frac{\Theta \in \mathsf{SDinat}(\mathsf{Hom}(F\ -, -), \mathsf{Hom}(C, -))}{\mathsf{build}_{F,C} \Theta : C \to \mu F}$$

We may ask now when a term $\Theta : \forall X.\ (FX \Rightarrow X) \Rightarrow C \Rightarrow X$ is guaranteed to be a strongly dinatural transformation. The answer depends on the specifics of the language and model under scrutiny and, in particular, on the semantics assigned to type quantification. This is a subtle and technical topic that we wish to discuss separately and in detail elsewhere. For this paper, we note however that in parametric models of polymorphic languages terms of type $\forall X.\ (FX \Rightarrow X) \Rightarrow C \Rightarrow X$ define strong dinaturals. From a practical point of view, it is then possible to get strong dinaturality for free.

We finish the discussion of BUILD by arguing that it makes perfect sense to separate language-specific questions such as when a term is strong dinatural from questions about basic semantic constructions appropriate for modelling some general phenomenon, e.g. the question of what BUILD is. We do not, for example, ask when a type constructor is a functor in a discussion of the initial algebra semantics. Certainly there exist simple sufficient syntactic conditions for functoriality, e.g., positivity, but the definition of any such condition and the proof of its sufficiency is specific to the language and model used. The situation with strong dinaturality of a term is completely analogous.

3 Augmenting build

BUILD presentations of producers do not always suffice for deforestation. Consider the definition of the `append` function for lists as a `foldr`:

```
append as bs = foldr (:) bs as
```

Abstracting the constructors, we get the following attempt of a definition of `append` in terms of `build`:

```
append as bs = build theta
    where theta c n = foldr c bs as
```

Unfortunately however, this attempt is incorrect, as the type of `theta` is not general enough. The problem is that the constructor occurrences in the list `bs`, while being part of the final result of `append`, are not abstracted in `theta`. So, one solution for the problem would be to replace the list `bs` with a traversal which uniformly replaces its constructor occurrences:

```
append as bs = build theta
    where theta c n = foldr c (foldr c n bs) as
```

Although this is indeed a correct definition of `append`, it introduces an extra traversal of the list `bs` and there is no guarantee that this traversal can be removed by subsequent fusion.

As a better solution, Gill [11] introduced a new combinator `augment` which generalizes `build` by abstracting out the nil-constructor position in a list:

```
augment :: (forall x. (a -> x -> x) -> x -> x) -> [a] -> [a]
augment theta bs = theta (:) bs
```

In terms of `augment`, `append` is easily expressed as follows:

```
append as bs = augment (\ c n -> foldr c n as) bs
```

Note, that `build` is just a special case of `augment`:

```
build theta == augment theta []
```

and `foldr/build` fusion or shortcut deforestation for lists generalizes to

```
foldr c n (augment theta bs) == theta c (foldr c n bs)
```

The same idea can be easily extended to arbitrary inductive types by abstracting over non-recursive constructors, to obtain a version of shortcut deforestation tailored specifically for functions that graft. This has been done by Johann [15]. We present however a slightly more general version and, following our general methodology, derive it from a unique existence situation.

Let $H : \mathcal{C} \to \mathcal{C}$ be a functor and let $T' : \mathcal{C} \to [\mathcal{C}, \mathcal{C}]$ be the functor given by $T'AX =_{\mathrm{df}} A + HX$. Then, if an initial $T'A$-algebra (= a free H-algebra over A) exists for every object A of $\mathcal{C}$, we can get a functor $T : \mathcal{C} \to \mathcal{C}$ by defining $TA =_{\mathrm{df}} \mu(T'A)$. This models an inductive type parameterized in non-recursive constructors. Decompose, each map $\mathsf{in}_{T'A}$ into two maps $\eta_A : A \to TA$ and $\tau_A : H(TA) \to TA$ by setting

$$\eta_A =_{\mathrm{df}} \mathsf{in}_{T'A} \circ \mathsf{inl}_{A,H(TA)}$$
$$\tau_A =_{\mathrm{df}} \mathsf{in}_{T'A} \circ \mathsf{inr}_{A,H(TA)}$$

Define, finally, for any map $f : A \to TB$, a map $f^\star : TA \to TB$ by

$$f^\star =_{\mathrm{df}} \mathsf{fold}_{T'A,TB}[\, f, \tau_B \,]$$

Conceptually, η packages the non-recursive constructors of the parameterized type, τ packages the recursive constructors, and $()^\star$ is substitution for non-recursive constructors. It is standard knowledge the data $(T, \eta, ()^\star)$ so constructed constitute a monad which, more specifically, is the free monad over H, but we will not make deep use of this fact in this paper. With the stage set, we can now proceed to a proposition which supplies the parameterized inductive type T with an AUGMENT combinator.

Proposition 4. *Let $\mathcal{C}$ be a category, $H : \mathcal{C} \to \mathcal{C}$ a functor such that initial algebras of all functors $T'A =_{\mathrm{df}} A + H-$ exist. Let T, η, τ, $()^\star$ be defined as above. Then, for any map $f : A \to TB$ and $U_{T'A}$-cone (C, Θ) there exists a unique map $h : C \to TB$ such that, for any $T'B$-algebra $(X, [\,\varphi_0, \varphi_1\,])$, it holds that*

$$\mathsf{fold}_{T'B,X}[\,\varphi_0, \varphi_1\,] \circ h = \Theta_X\, ([\, \mathsf{fold}_{T'B,X}[\,\varphi_0, \varphi_1\,] \circ f, \varphi_1\,])$$

We denote the unique map by $\mathsf{augment}_{T',C}(\Theta, f)$.

Proof. We prove that

$$\mathsf{augment}_{T',C}(\Theta, f) = \Theta_{TB}[\, f, \tau_B\,]$$

It is easy to see that $\mathsf{fold}_{T'B,X}[\,\varphi_0, \varphi_1\,] : TB \to X$ is a $T'A$-algebra map from $(TB, [\, f, \tau_B\,])$ to $(X, [\, \mathsf{fold}_{T'B,X}[\,\varphi_0, \varphi_1\,] \circ f, \varphi_1\,])$. Hence from (C, Θ) being a $U_{T'A}$-cone,

$$\mathsf{fold}_{T'B,X}[\,\varphi_0, \varphi_1\,] \circ \Theta_{TB}[\, f, \tau_B\,] = \Theta_X[\, \mathsf{fold}_{T'B,X}[\,\varphi_0, \varphi_1\,] \circ f, \varphi_1\,]$$

as needed.

Assume now that there is a map $h : C \to TB$ such that, for any $T'B$-algebra $(X, [\,\varphi_0, \varphi_1\,])$, it holds that

$$\mathsf{fold}_{T'B,X}[\,\varphi_0, \varphi_1\,] \circ h = \Theta_X\,[\,\mathsf{fold}_{T'B,X}[\,\varphi_0, \varphi_1\,] \circ f, \varphi_1\,]$$

Then

$$\begin{aligned} h &= \mathsf{fold}_{T'B,TB}[\,\eta_B, \tau_B\,] \circ h \\ &= \Theta_{TB}\,[\,\mathsf{fold}_{T'B,X}[\,\eta_B, \tau_B\,] \circ f, \tau_B\,] \\ &= \Theta_{TB}\,[\,f, \tau_B\,] \end{aligned}$$

so we also have uniqueness. □

On the level of syntax, the proposition proved justifies the introduction of a combinator $\mathsf{augment}_{T'}$ with a typing rule

$$\frac{(C, \Theta) \in U_{T'A}\text{-}\mathbf{cone} \quad f : A \to TB}{\mathsf{augment}_{T',C}(\Theta, f) : C \to TB}$$

and β-conversion rule

$$\frac{(C, \Theta) \in U_{T'A}\text{-}\mathbf{cone} \quad f : A \to TB \quad (X, [\,\varphi_0, \varphi_1\,]) \in T'B\text{-}\mathbf{alg}}{\mathsf{fold}_{T'B,X}[\,\varphi_0, \varphi_1\,] \circ \mathsf{augment}_{T',C}(\Theta, f) = \Theta_X[\,\mathsf{fold}_{T'B,X}[\,\varphi_0, \varphi_1\,] \circ f, \varphi_1\,]}$$

which is fusion of FOLD and AUGMENT. One also gets the following conversion rules relating to the monad structure on T:

$$\frac{(C, \Theta) \in U_{T'A}\text{-}\mathbf{cone} \quad f : A \to TB}{\mathsf{augment}_{T',C}(\Theta, f) = f^\star \circ \mathsf{build}_{T'A,C}\Theta} \qquad \frac{(C, \Theta) \in U_{T'A}\text{-}\mathbf{cone}}{\mathsf{build}_{T'A,C}\Theta = \mathsf{augment}_{T',C}(\Theta, \eta_A)}$$

$$\frac{(C, \Theta) \in U_{T'A}\text{-}\mathbf{cone} \quad f : A \to TB \quad g : B \to TC}{g^\star \circ \mathsf{augment}_{T',C}(\Theta, f) = \mathsf{augment}_{T',C}(\Theta, g^\star \circ f)}$$

Johann [15] describes essentially the same combinator, but in a restricted form where the type of non-recursive constructors is fixed (so that $B = A$). (The reason must be that, in the prototypical case of lists, the type of non-recursive constructors is constantly 1; normally, one does not consider the possibility of supporting multiple nil's drawn from a parameter type.) Unfortunately, this restriction hides the rather important role of the monad structure on T.

As an example, we may consider the parameterized type of binary leaf labelled trees, Haskell-implementable as follows:

```
data BLTree a = Leaf a | Bin (BLTree a) (BLTree a)

foldB :: (a -> x) -> (x -> x -> x) -> BLTree a -> x
foldB l b (Leaf a) = l a
foldB l b (Bin as0 as1) = b (foldB l b as0) (foldB l b as1)
```

The BUILD and AUGMENT combinators are implementable as follows:

```
buildB :: (forall x. (a -> x) -> (x -> x -> x) -> x) -> BLTree a
buildB theta = theta Leaf Bin

augmentB :: (forall x. (a -> x) -> (x -> x -> x) -> x)
                                  -> (a -> BLTree b) -> BLTree b
augmentB theta f = theta f Bin
```

The shortcut deforestation laws say that

```
foldB l b (buildB theta) == theta l b

foldB l b (augmentB theta f) == theta (foldB l b . f) b
```

Elsewhere we will show that there is no reason to stop at AUGMENT for free monads. A similar combinator is possible for any monad obtained from a parameterized monad via initial algebras as described in [32]. Some parameterized inductive types covered by this more general construction are, e.g., finitely branching node labelled trees and inductive hyperfunctions [17].

4 Coinductive types and destroy

Since coinductive types are dual to inductive types, it is clear that, if inductive types admit a universal characterization with BUILDS as the mediating maps, then there must be a universal characterization of coinductive types centered around a dual combinator. We now turn to this characterization.

In the standard modelling, a coinductive type is a chosen final coalgebra of a functor.

Definition 5 ((final) coalgebra). *Let $\mathcal{C}$ be a category and $F : \mathcal{C} \to \mathcal{C}$ be a functor. An* F-coalgebra *is an object X together with a map $\varphi : X \to F\ X$. An* F-coalgebra map $(X, \varphi) \to (Y, \psi)$ *is a map $f : X \to Y$ such that $\psi \circ f = F f \circ \varphi$. A* final F-coalgebra *is a final object in the category F-**coalg** of F-coalgebras, i.e., an F-coalgebra with a unique map to it from any F-coalgebra.*

We agree to denote a chosen final F-coalgebra by $(\nu F, \mathsf{out}_F)$, and the unique map to it from an F-coalgebra (X, φ) by $\mathsf{unfold}_{F,X}\varphi$. The final coalgebra semantics justifies the well-known syntax of coinductive types which is given by a type νF, a destructor out_F and a constructor unfold_F subjected to the following typing and conversion rules:

– typing rules:

$$\mathsf{out}_F : \nu F \to F(\nu F) \quad \frac{(X, \varphi) \in F\text{-}\mathbf{coalg}}{\mathsf{unfold}_{F,X}\varphi : X \to \nu F}$$

– β-conversion rule (= cancellation law for UNFOLD):

$$\frac{(X,\varphi) \in F\text{-}\mathbf{coalg}}{\mathsf{out}_F \circ \mathsf{unfold}_{F,X}\varphi = F(\mathsf{unfold}_{F,X}\varphi) \circ \varphi}$$

– η- and permutative conversion rules (= identity and fusion laws for UNFOLD):

$$\mathsf{id}_{\nu F} = \mathsf{unfold}_{F,F(\nu F)}\mathsf{out}_F \qquad \frac{f : (X,\varphi) \to (Y,\psi) \in F\text{-}\mathbf{coalg}}{\mathsf{unfold}_{F,Y}\psi \circ f = \mathsf{unfold}_{F,X}\varphi}$$

To give some examples, the type of streams over a given type A can be modelled by a final coalgebra of functor $A \times -$, the type of colists (possibly infinite lists) over A by a final coalgebra of the functor $1 + A \times -$ etc.

A Haskell implementation of the final coalgebra view of streams based on the β-conversion rule would be the following:

```
data Str a = forall x . UnfoldS (x -> a) (x -> x) x

hdS :: Str a -> a
hdS (UnfoldS h t x) = h x

tlS :: Str a -> Str a
tlS (UnfoldS h t x) = UnfoldS h t (t x)
```

The usual implementation of streams, which avoids rank-2 type signatures, exploits the fact that the final coalgebra structure map is an isomorphism.

```
data Str a = MkStr { hd :: a, tl :: Str a }

unfoldS :: (x -> a) -> (x -> x) -> x -> Str a
unfoldS h t x = MkStr { hd = h x, tl = unfoldS h t (t x) }
```

Colists are Haskell-implemented, e.g., by Haskell's (lazy) lists and `unfoldr`:

```
unfoldr ::  (x -> Maybe (a, x)) -> x -> [a]
unfoldr phi x = case phi x of
                  Nothing     -> []
                  Just (a,x') -> a : unfoldr phi x'
```

(Recall that Haskell is semantically based on **CPO** where inductive and coinductive types coincide.)

Our alternative modelling of the coinductive type given by a functor F views it as a colimit of the forgetful functor $V_F : F\text{-}\mathbf{coalg} \to \mathcal{C}$. We recall the definition of a colimit of a functor.

Definition 6 (colimit). *A* cocone *for a functor $J : \mathcal{C} \to \mathcal{D}$ is a pair (C, γ) consisting of an object C in $\mathcal{D}$ and, for each object X in $\mathcal{C}$, a map $\gamma_X : JX \to C$ in $\mathcal{D}$ such that, for every map $f : X \to Y$ in $\mathcal{C}$, we have $\gamma_Y \circ Jf = \gamma_X$. A* map of cocones *$(C,\gamma) \to (D,\delta)$ is a map $h : C \to D$ in $\mathcal{D}$ such that, for each object X in $\mathcal{C}$, we have $h \circ \gamma_X = \delta_X$. A* colimit *of J is an initial object in the category of J-**cocone** of J-cocones.*

By this definition, a cocone of V_F is given by an object C and, for any F-coalgebra (X, φ), a map $\Theta_X\ \varphi : X \to C$, such that, for any F-coalgebra morphism $f : (X, \varphi) \to (Y, \psi)$, we have $\Theta_Y\ \psi \circ f = \Theta_X\ \varphi$ (which is to say that Θ is a strongly dinatural transformation from $\mathsf{Hom}(-, F\ -)$ to $\mathsf{Hom}(-, C)$). A V_F-cocone map $(C, \Theta) \to (D, \Xi)$ is a map $h : C \to D$ such that, for any F-coalgebra (X, φ), we have $h \circ \Theta_X\ \varphi = \Xi_X\ \varphi$. A V_F-colimit is a V_F-cocone with a unique map to every V_F-cocone. We denote a chosen V_F-colimit by $(\nu^* F, \mathsf{unfold}^*_F)$ and the unique V_F-cocone map from a given V_F-cocone (C, Θ) by $\mathsf{destroy}^*_{F,C}\Theta$. Intuitively, a V_F-cocone is an object with a fuseable unfold-like operation and the V_F-limit is the smallest such.

The syntax derived from the alternative semantics consists of a type $\nu^* F$ along with a destructor unfold_F and constructor $\mathsf{destroy}_F$ plus these rules:

– typing rules:

$$\frac{(X, \varphi) \in F\text{-}\mathbf{coalg}}{\mathsf{unfold}^*_{F,X}\ \varphi : C \to \nu^* F} \qquad \frac{f : (X, \varphi) \to (Y, \psi) \in F\text{-}\mathbf{coalg}}{\mathsf{unfold}^*_{F,Y}\psi \circ f = \mathsf{unfold}^*_{F,X}\varphi} \qquad \frac{(C, \Theta) \in V_F\text{-}\mathbf{cocone}}{\mathsf{destroy}^*_{F,C}\Theta : \nu^* F \to C}$$

– β-conversion rule:

$$\frac{(C, \Theta) \in V_F\text{-}\mathbf{cocone} \quad (X, \varphi) \in F\text{-}\mathbf{coalg}}{\mathsf{destroy}^*_{F,C}\Theta \circ \mathsf{unfold}^*_{F,X}\ \varphi = \Theta_X\ \varphi}$$

– η- and permutative conversion rules:

$$\mathsf{id}_{\nu^* F} = \mathsf{destroy}^*_{F,\nu^* F}\mathsf{unfold}^*_F \qquad \frac{h : (C, \Theta) \to (D, \Xi) \in V_F\text{-}\mathbf{cocone}}{h \circ \mathsf{destroy}^*_{F,C}\Theta = \mathsf{destroy}^*_{F,D}\Xi}$$

The equivalence of the two semantics is established by the following proposition dualizing Propositions 1, 2.

Proposition 5. *Let $\mathcal{C}$ be a category and $F : \mathcal{C} \to \mathcal{C}$ a functor. Then, (a) if there is a final F-coalgebra $(\nu F, \mathsf{out}_F)$, then νF is the vertex of a V_F-colimit, and (b) if there is a V_F-colimit $(\nu^* F, \mathsf{unfold}^*_F)$, then $\nu^* F$ is the carrier of a final F-coalgebra.*

Proof (constructions). The statements are immediate by duality from Propositions 1, 2. But to show the constructions, we sketch an explicit dual proof.

(a) Set, for any V_F-cocone (C, Θ), $\mathsf{destroy}_{F,C}\Theta =_{\mathrm{df}} \Theta_{\nu F}\ \mathsf{out}_F$. Check that $(\nu F, \mathsf{unfold}_F, \mathsf{destroy}_F)$ is a V_F-colimit.

(b) Set, for any F-coalgebra (X, φ), $\mathsf{unfoldout}^*_{F,X}\ \varphi =_{\mathrm{df}} F\ \mathsf{unfold}^*_{F,X}\ \varphi \circ \varphi$. Set $\mathsf{out}^*_F =_{\mathrm{df}} \mathsf{destroy}^*_{F,F\,(\nu^* F)}\mathsf{unfoldout}^*_F$. Check that $(\nu^* F, \mathsf{out}^*_F, \mathsf{unfold}^*_F)$ is a final F-coalgebra. □

From (a), we get a new combinator for coinductive types, DESTROY, and a new shortcut deforestation law: the DESTROY combinator is derived from a final coalgebra being a colimit of the functor forgetting coalgebra structure, and the shortcut deforestation or UNFOLD/DESTROY fusion law is just the corresponding β-conversion rule. For streams, in particular, we get a DESTROY combinator with the following Haskell implementation:

```
destroyS :: (forall x . (x -> a) -> (x -> x) -> x -> c) -> Str a -> c
destroyS theta = theta hdS tlS
```

For possibly infinite lists, we obtain the following combinator:

```
destroyL :: (forall x . (x -> Maybe (a, x)) -> x -> c) -> [a] -> c
destroyL theta = theta OutL
         where OutL [] = Nothing
               OutL (a : as) = Just (a, as)
```

Shortcut deforestation for streams says

```
destroyS theta . unfoldS h t == theta h t
```

while for colists it says

```
destroyL theta . unfoldr phi == theta phi
```

The DESTROY combinator and shortcut deforestation for colists appear in Gill [11] and Svenningsson [29]. In fact, they speak of Haskell's lists, but to have the transformation correct, it is essential that the type is coinductive.

Part (b) of the proposition gives an alternative arrangement of syntax for programming with coinductive types where DESTROY rather than OUT is primitive. For streams, a possible complete Haskell implementation is this:

```
data Str a = forall x . UnfoldS (x -> a) (x -> x) x

destroyS :: (forall x . (x -> a) -> (x -> x) -> x -> c) -> Str a -> c
destroyS theta (UnfoldS h t x) = theta h t x
```

The usual head and tail destructors, which constitute the OUT destructor for streams, are non-primitive in this implementation. They are definable functions:

```
hdS :: Str a -> a
hdS = destroyS (\ h _ -> h)

tlS :: Str a -> Str a
tlS = destroyS (\ h t -> UnfoldS h t . t)
```

As an example, we could program stream indexing as follows:

```
fromS :: Int -> Str a -> a
fromS n = destroyS (\ h t -> h . ntimes n t)
```

We find that the terms $\mathsf{fromS}\, n$ deserve to be called *Church indexicals*.

Just as the BUILD combinator for inductive types admits a generalization to an AUGMENT combinator for monadic parameterized inductive types, a dual generalization of DESTROY is possible for comonadic parameterized coinductive types. We will not spell out the details here.

5 Related work

Programming and reasoning about inductive and coinductive types with combinators derived from the initial algebra and final coalgebra semantics was first explored by Hagino [13]. To the functional programming community, this idea was introduced in [21, 22, 28] and became very popular then. The method to encode inductive types via type quantification (the "impredicative encoding") was first described in [19, 3]. Shortcut deforestation for lists was proposed by Gill et al. [11, 10]. For general inductive types, it was defined in [30, 18].

The landmark works on parametricity are Reynold's and Wadler's papers [26, 27, 34], categorical studies on the semantics of polymorphism based on dinaturality and strengthenings include [12, 2, 8]. Some discussions of the implications of parametricity for (co)inductive types are [36, 14, 1]. Strong dinaturality, most probably first introduced in [23], has recently been studied closely in [5, 6].

Most closely related to the work reported here, Johann [16, 15] has given a thorough proof of the correctness of FOLD/BUILD and FOLD/AUGMENT fusion via parametricity of contextual equivalence. Pavlovic [25] has analyzed FOLD/BUILD fusion in terms of "paranatural" transformations which are essentially the same as strong dinaturals.

6 Conclusions

We have shown that besides the initial algebra semantics inductive types admit an alternative but equivalent semantics in terms of limits of forgetful functors. This equivalent semantics matches nicely the FOLD/BUILD syntax that has been invented for the purpose of program transformations. In particular, it gives a language-independent axiomatic specification of BUILD where the correctness of shortcut deforestation is an axiom. This separation between the general semantics of inductive types and parametricity theorems for specific languages and their models is, to our view, good and helpful, because of the modularity and clarity it brings.

As future work we intend to give an account of rational types (essentially non-wellfounded trees with finitely many distinct subtrees) based on the general theory put forward in Ghani et al. [9] in order to systematically study disciplines for programming with rational types such as the "cycle therapy" of Turbak and Wells [31]. We also plan to achieve a similar account for the vanish combinators à la Voigtländer [33]. A further topic will be parametricity in terms of strong dinaturals for languages supporting interleaved inductive and coinductive types.

Acknowledgments The authors are thankful to Patricia Johann and John Launchbury for useful comments.

All three authors benefitted from the support from the EU FP5 IST programme via the APPSEM II project and from the Royal Society ESEP programme within joint research project No. 15642. The second and third author were also partially supported by the Estonian Science Foundation under grant No. 5567.

References

1. T. Altenkirch. Logical relations and inductive/coinductive types. In G. Gottlob, E. Grandjean, and K. Seyr, eds., *Proc. of 12th Int. Wksh. on Computer Science Logic, CSL'98*, v. 1584 of *Lecture Notes in Computer Science*, pp. 343–354. Springer-Verlag, 1999.
2. E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, 1990. Corrigendum, ibid., 71(3):431, 1991.
3. C. Böhm and A. Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39(2–3):135–154, 1985.
4. E. Dubuc and R. Street. Dinatural transformations. In S. M. Lane, ed., *Reports of the Midwest Category Seminar IV*, v. 137 of *Lecture Notes in Mathematics*, pp. 126–137. Springer-Verlag, 1970.
5. A. Eppendahl. Parametricity and Mulry's strong dinaturality. Technical Report 768, Dept. of Computer Science, Queen Mary and Westfield College, London, 1999.
6. A. Eppendahl. *Categories and Types for Axiomatic Domain Theory*. PhD thesis, Queen Mary, University of London, 2003.
7. L. Fegaras. Using the parametricity proposition for program fusion. Technical report CSE-96-001, Dept. of Computer Science and Engineering, Oregon Graduate Institute, Portland, OR, 1996.
8. P. J. Freyd. Structural polymorphism. *Theoretical Computer Science*, 115(1):107–129, 1993.
9. N. Ghani, C. Lüth, and F. D. Marchi. Coalgebraic monads. In L. S. Moss, ed., *Proc. of 5th Wksh. on Coalgebraic Methods in Computer Science, CMCS'02*, v. 65(1) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.
10. A. Gill, J. Launchbury, and S. L. P. Jones. A short cut to deforestation. In *Conf. Record of 6th ACM SIGPLAN-SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'93*, pp. 223–232. ACM Press, 1993.
11. A. J. Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Univ. of Glasgow, 1996.
12. J.-Y. Girard, A. Scedrov, and P. J. Scott. Normal forms and cut-free proofs as natural transformations. In Y. N. Moschovakis, ed., *Logic from Computer Science*, v. 21 of *Mathematical Sciences Research Institute Publications*, pp. 217–241. Springer-Verlag, 1991.
13. T. Hagino. A typed lambda calculus with categorical type constructors. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, eds., *Proc. of 2nd Int. Conf. on Category Theory and Computer Science, CTCS'87*, v. 283 of *Lecture Notes in Computer Science*, pp. 140–157. Springer-Verlag, 1987.
14. R. Hasegawa. Categorical data types in parametric polymorphism. *Mathematical Structures in Computer Science*, 4(1):71–109, 1994.
15. P. Johann. A generalization of short-cut fusion and its correctness proof. *Higher-Order and Symbolic Computation*, 15(4):273–300, 2002.
16. P. Johann. Short-cut fusion is correct. *Journal of Functional Programming*, 13(4):797–814, 2003.
17. S. Krstić, J. Launchbury, and D. Pavlović. Categories of processes enriched in final coalgebras. In F. Honsell and M. Miculan, eds., *Proc. of 4th Int. Conf. on Found. of Software Science and Computation Structures, FoSSaCS'01*, v. 2030 of *Lecture Notes in Computer Science*, pp. 303–317. Springer-Verlag, 2001.

18. J. Launchbury and T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Conf. Record 7th ACM SIGPLAN-SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95*, pp. 314–323. ACM Press, 1995.
19. D. Leivant. Reasoning about functional programs and complexity classes associated with type disciplines. In *Proc. of 24th Annual IEEE Symp. on Foundations of Computer Science, FOCS'83*, pp. 460–469. IEEE CS Press, 1983.
20. S. Mac Lane. *Categories for the Working Mathematician*, v. 5 of *Graduate Texts in Mathematics*, 2nd ed. Springer-Verlag, 1997. (1st ed., 1971).
21. G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–279, 1990.
22. E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, ed., *Proc. of 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'91*, v. 523 of *Lecture Notes in Computer Science*, pp. 124–144. Springer-Verlag, 1991.
23. P. S. Mulry. Strong monads, algebras and fixed points. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, eds., *Applications of Categories in Computer Science*, v. 177 of *London Math. Society Lecture Note Series*, pp. 202–216. Cambridge University Press, 1992.
24. R. Paré and L. Román. Dinatural numbers. *Journal of Pure and Applied Algebra*, 128(1):33–92, 1998.
25. D. Pavlovic. Logic of build fusion. Technical Report KES.U.00.9, Kestrel Institute, 2000.
26. J. C. Reynolds. Towards a theory of type structure. In B. Robinet, ed., *Proc. of Programming Symp. (Colloque sur la programmation)*, v. 19 of *Lecture Notes in Computer Science*, pp. 408–425. Springer-Verlag, 1974.
27. J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, ed., *Proc. of 9th IFIP World Computer Congress, Information Processing '83*, pp. 513–523. North-Holland, 1983.
28. T. Sheard and L. Fegaras. A fold for all seasons. In *Proc. of 6th ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'93*, pp. 233–242. ACM Press, 1993.
29. J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proc. of 7th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'02*, v. 37(9) of *SIGPLAN Notices*, pp. 124–132. ACM Press, 2002.
30. A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Conf. Record 7th ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95*, pp. 306–313. ACM Press, 1995.
31. F. Turbak and J. B. Wells. Cycle therapy: A prescription for fold and unfold on regular trees. In *Proc. of 3rd Int. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming, PPDP'01*, pp. 137–149. ACM Press, 2001.
32. T. Uustalu. Generalizing substitution. *Theoretical Informatics and Applications*, 37(4):315-336, 2003.
33. J. Voigtländer. Concatenate, reverse and map vanish for free. In *Proc. of 7th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'02*, v. 37(9) of *SIGPLAN Notices*, pp. 14–25. ACM Press, 2002.
34. P. Wadler. Theorems for free! In *Proc. of 4th Int. Conf. on Funct. Prog. Languages and Computer Arch., FPCA'89*, pp. 347–359. ACM Press, 1989.
35. P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.
36. P. Wadler. Recursive types for free! Draft manuscript, 1990.